**Efficient Synchronization**
**On Multiprocessors With Shared Memory**

by

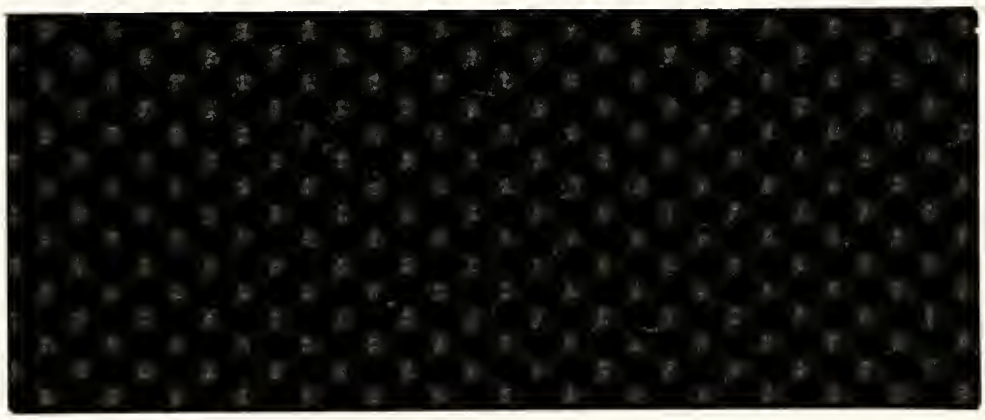
*Clyde P. Kruskal†*
*Larry Rudolph‡*
*Marc Snir‡*

Ultracomputer Note #105
May, 1986

Ultracomputer Research Laboratory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012

C.1



**Efficient Synchronization**
**On Multiprocessors With Shared Memory**

by

*Clyde P. Kruskal*†
*Larry Rudolph*‡
*Marc Snir*‡

Ultracomputer Note #105
May, 1986

†Computer Science Department University of Maryland College Park, Maryland

‡Institute of Mathematics and Computer Science The Hebrew University of Jerusalem Jerusalem, Israel

Part of the work was done while the first author was at the University of Illinois, the second author was at Carnegie-Mellon University, and the third author was at the Courant Institute of Mathematical Sciences, New York University.

To appear in 5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 1986.

*ABSTRACT*

A new formalism is given for read-modify-write (RMW) synchronization operations. This formalism is used to extend the memory reference combining mechanism, introduced in the NYU Ultracomputer, to arbitrary RMW operations. A formal correctness proof of this combining mechanism is given. General requirements for the practicality of combining are discussed. Combining is shown to be practical for many useful memory access operations. This includes memory updates of the form *mem_val := mem_val op val*, where op need not be associative, and a variety of synchronization primitives. The computation involved is shown to be closely related to parallel prefix evaluation.

## 1. Introduction

Shared memory provides convenient communication between processes in a tightly coupled multiprocessing system. Shared variables can be used for data sharing, information transfer between processes, and, in particular, for coordination and synchronization. Constructs such as the semaphore introduced by Dijkstra in [Di], and the many variants that followed, provide convenient solutions to many synchronization problems involving arbitrary number of processes. These constructs are supported in hardware by machine instructions that atomically execute a Read-Modify-Write cycle. Such instructions exist on most modern CPU's.

An atomic Read-Modify-Write operation only requires that it be semantically atomic, although it is often processed atomically also. The "serial bottleneck" created by this atomic processing, while acceptable for small scale parallelism, can seriously impair the performance of a system with thousands of processors.

Frequent accesses to a shared variable not only slow down those processes performing the access, but may cause the entire machine to thrash. Large-scale shared memory parallel processors are likely to use multistage packet switched interconnection networks for processor to memory traffic. These networks provide high bandwidth and short latency time when memory accesses are distributed randomly, but, if even a small percentage of the memory requests are directed to one specific spot, the network becomes congested and performance quickly degrades. A recent study of Pfister and Norton [PN] shows that not only those processors attempting to access the same "hot spot" are delayed, but also the remaining processors. Although replication of data can often be used to circumvent the hot spot problem for read-only data, it cannot be used for synchronization variables.

The performance degradation can be mitigated by a memory request "combining" technique (which will be described later). Briefly, combining works as follows: When a "conflict" occurs within the network for the same switch output port for memory requests directed to the same location, a new combined request that represents the conflicting requests is created. Separate replies to the original requests are later created from the reply to the combined request. The logic for combining and uncombining memory references is distributed throughout the processor to memory interconnection network.

It is worthwhile emphasizing that such simultaneous requests directed at the same memory cell are not random, rare events. When processed in an efficient manner, they can form the basis for a completely parallel, decentralized operating system as well as a building block for efficient parallel programming constructs. A general discussion of the cost/performance tradeoffs of the combining mechanism has been argued elsewhere.

Indeed, such a mechanism was proposed for read requests in the CHoPP machine [SBK]. It was extended to handle write requests, and some types of Read-Modify-Write requests [Ru] and further generalized for associative Read-Modify-Write operations [GK]. These ideas are used to implement concurrent reads, writes, and "Fetch-and-Adds" in the NYU Ultracomputer [GGK] and IBM RP3 [PBH] machines.

The semantics of serial processes are well understood; it is relatively easy to argue on the correctness of serial computers. The situation is quite different for parallel systems: Satisfactory definitions of their semantics have only recently evolved ([LyF], [La1], [La2], [La3]) and our intuition often fails when trying to formally reason about parallel systems. Therefore, it is important to precisely define correctness criteria for parallel systems and to formally argue that these criteria are fulfilled.

We show that combining fulfills two important criteria: (1) Combining is a general technique that applies to arbitrary memory access operations, not just an *ad hoc* method to handle the NYU Ultracomputer operations. (2) This new interconnect mechanism does not change the properties of the processor-memory system.

In this paper we address these issues rigorously. A new, very general formalism for read-modify-write (RMW) operations is given. A general definition is given of a correct machine implementation. A method for combining general RMW operations is given and proven to be correct. Several families of memory access operations are analyzed using this

general framework. This includes familiar operations such as load, store, swap, test-and-set, fetch-and-add, and general data-level synchronization primitives (see [GP]). It is well known that any associative operation can be combined efficiently [GK]. We show that other combinable families of operations include the four standard arithmetic operations, all sixteen boolean functions, and synchronization methods such as full/empty bits. Implementation issues concerning support of such primitives are considered. Finally, the combining mechanism is shown to be closely related to the parallel prefix computation problem [LaF].

## 2. Read-Modify-Write

We use a formalism similar to that developed by Lynch and Fisher [LyF]: A parallel computation consists of a set of processes that execute in parallel. Each of these processes is considered to be a sequential program augmented with the ability to access global, shared variables. We restrict our attention to shared memory access techniques and assume standard operations for manipulation of local (or private) data.

Instead of the usual load and store memory access operations of sequential processing, all accesses to shared variables are assumed to be *Read-Modify-Write (RMW)* operations. The operation (or instruction) RMW(X,f), where X is a shared variable and *f* is a mapping, is defined to be equivalent to the indivisible execution of the following function:

```
function RMW(X,f)
  begin
     temp ← X;
     X ← f(X);
     return(temp)
  end
```

This operation yields, as its value, the old value of the variable X and also updates the value stored in X according to the updating transformation *f*.

The usual load and store operations are particular cases of RMW operations: a load from (the address of) variable X is equivalent to RMW(X,**id**), where **id** is the identity mapping (i.e. $f(x) = x$). A store of value v to variable X is equivalent to $RMW(X,I_v)$, where $I_v$ is the mapping that has constant value v (i.e. $f(x) = v$); the returned value is ignored. In fact, an assignment of the form $Y \leftarrow RMW(X,I_Y)$, where Y is a private variable and X is a shared variable, implements a *swap* instruction: X and Y swap values. Note that the usual use of swap instructions is to exchange values between a shared

variable (the lock) and a private variable (the key) (see, e.g. [PET], §9.5.4).

The well known *test-and-set* instruction can also be implemented as an RMW instruction. We have

$$\text{test-and-set}(X) \equiv \text{RMW}(X, I_{true}) .$$

A more powerful RMW operation is the *fetch-and-add* synchronization primitive. It is defined by

$$\text{fetch-and-add}(X,a) \equiv \text{RMW}(X, +_a) ,$$

where $+_a$ is Curried addition, i.e. $+_a(x) = x+a$. It corresponds to the indivisible execution of the following code.

```
function fetch-and-add(X,a)
  begin
     temp ← X;
     X ← X+a;
     return(temp)
  end
```

A similar operation (replace-add) was introduced many years ago [DGSS]. It was independently considered by Dijkstra [Di] who rejected it, believing it to be an inadequate tool for synchronization. It nevertheless turned out to be a very useful synchronization primitive, and was essential in the development of efficient coordination code for the NYU Ultracomputer operating system [Ru],[GLR]. The change from replace-add to fetch-and-add [GK] simplified the combining logic and paved the way to the general result given in this paper.

Any memory access that consists of reading one shared memory location, performing an arbitrary local computation, then updating the memory location can be expressed as an RMW operation of the above form. This is the general form for memory accesses assumed by [LyF], and seems to encompass most, if not all, useful synchronization operations based on shared variables. Other examples of RMW operations will be presented in later sections.

## 3. Semantics

In their classic paper describing the IBM 360 system, Amdahl, Blaauw, and Brooks [ABB] introduced the notions of architecture, implementation, and realization. The architecture can be thought of as the abstract machine that is presented to the user at the assembly language level or presented in the principles of operations manual. The implementation is how hardware is used to implement the features and operations of the architecture. The realization is the exact specification of the hardware, such as which chips are used and how they are wired together. In an implementation, each "atomic" operation of the architecture may actually consist of several "subatomic" microoperations; the implementation may use stores[1] that are not visible to the user. The implementation is correct if its *visible* behavior is a correct behavior of the architecture: the initial to final state mapping on visible stores is the same for the architecture as for the implementation. A similar situation holds for the realization. These definitions can be extended and generalized to all the levels of an architecture, software and hardware. At each level an architecture is implemented by a lower one; the implementation is correct if it yields the same visible behavior.

### 3.1. Definitions

We use a formalism similar to that developed by Lamport [La1],[La3]. The state of a machine is represented by the values of its stores. There are *stable* stores, such as memory, registers, status flags, etc., and *transient* stores, such as messages. Stable stores support nondestructive read and write operations. Messages are created by message transmission operations, and destroyed by message reception operations. They are used for internal communication and communication with the external world (I/O messages).

The execution of the computer can be viewed as consisting of a set of *atomic events*. Each atomic event may modify the value of one or more stores, and create or receive one or more messages. The semantics of an atomic event is defined by a mapping that specifies the state transformation associated with it: messages consumed, messages created and their values, and new values of modified stores. This naturally extends to a definition of the semantics of a sequence of atomic events by composition of mappings: in a sequence, event i+1 produces a new state based on the state produced by event i.

[1] In this section and the next, the term *store* will denote the state information; the term *write* will denote

We assume that one can observe the initial contents of the stores, the final content of the stores, and the order of I/O events (input reception and output transmission) as well as their values. That is, the observable behavior of a system consists of the (i) initial state to final state mapping induced by the computation and (ii) the sequence of I/O events occurring during the computation. Since we can observe the time (or order) of each external communication event, we can consider them to be totally ordered.

Many atomic events may occur concurrently; the order of occurrence of two events is significant only if their execution order affects the observable behavior of the system. This motivates the following definitions. Two sequences of events are *equivalent* if for any initial value of the stores and any sequence of input messages, the execution of these two sequences yield the same final values of the visible stores and the same sequence of I/O events. A *system execution* is a set of events partially ordered by a relation $\rightarrow$ such that any two extensions of $\rightarrow$ to total orders yield equivalent sequences of events. We say that event $\alpha$ *precedes* event $\beta$ if $\alpha \rightarrow \beta$. Our definition implies that the execution order $\rightarrow$ captures all dependencies that exist between atomic events.

The definition of a system execution (usually) implies that the relation $\rightarrow$ has the following properties:

(1) If u and v access the same store, and one of the accesses is a write access then either $u \rightarrow v$ or $v \rightarrow u$. (An event "reads" the stores that are in the domain of the mapping associated with it, and "writes" the stores that are in the range of this mapping.)

(2) If u and v are external communication events and u occurs before v then $u \rightarrow v$ (remember that external communication events are totally ordered).

(3) If u creates a message and v receives that message then $u \rightarrow v$.

The architecture of a computer is understood in terms of *operations*; each operation may consist of several atomic events. The partial order relation $\rightarrow$ on atomic events induces a relation, denoted by $\in$, on operations. Operation u *precedes* operation v, i.e. $u \in v$, if some event of u precedes some event of v.

Correctness criteria are expressed in terms of constraints on possible system executions; a system is defined by the set of legal system executions. For example, if operations

---

the store operation.

are required to be *atomic* then the execution relation $\epsilon$ induced by $\rightarrow$ must be a partial order: A cycle $u\epsilon \cdots \epsilon u$ implies that some event, say $\beta$, of an operation, say v, can be seen to occur after some event, say $\alpha_1$, belonging to u, i.e. $(\alpha_1 \rightarrow \beta)$, and before some other event, say $\alpha_2$ belonging to u, i.e. $(\beta \rightarrow \alpha_2)$; this implies u is not indivisible. Conversely, if the execution relation induces a partial order on operations then it can be extended to a total order so that events belonging to the same operation are contiguous: The outcome of the execution is as if the operations were executed serially, with each operation terminating before the next one starts.

## 3.2. Composing Systems from Subsystems

It is often convenient to define a system as a composition of subsystems. The stores of the system are the stores of the subsystems and the events of the system are events of the subsystems. We assume that subsystems communicate only by messages: an event of a subsystem may modify only stores of that subsystem, but it may create a message that is latter consumed by another subsystem. (This is similar to the work pioneered by Milner and Milne [MM] in the context of synchronous communicating processes.)

The semantic specification of the global system is derived from the semantic specifications of the composing subsystems. Each event is associated with the corresponding mapping in its subsystem. The set of legal system executions is defined as follows:

Let $\rightarrow$ be partial order on events of the system. This order induces an ordering of the events within each subsystem. This partial order is not necessarily an execution order: the relation $\rightarrow$ may not define an order on communication events that are external to a subsystem but internal to the global system. When a subsystem is considered in isolation, the order in which it executes external communication events is deemed meaningful; when it is part of a bigger system the order of its communication with other subsystems may not be meaningful, i.e. does not necessarily affect the global behavior of the system.

The relation $\rightarrow$ defines a correct system execution if it can be extended (by ordering all communication events within each subsystem) to a relation $\Rightarrow$ , such that the restriction of $\Rightarrow$ to each subsystem is a correct execution of the subsystem. Informally, a global system execution is correct if each subsystem may view it as a correct local system execution, where these different views are consistent.

### 3.3. An Example – The Uniprocessor

For example, we can consider a serial computer to consist of two separate subsystems: processor and memory. The processor executes a stream of instructions. The memory accepts a stream of requests (read, write, read-modify-write, etc.). Each request may modify the memory content and return a value.

Assume we have a formal definition for a correct execution by a serial processor. Informally, such a definition associates with each instruction a sequence of memory accesses and a mapping that computes the next state of the processor, given the current state and the values returned from memory. It specifies that instructions are executed atomically, so that the outcome of the execution of a stream of instructions can be computed by composing the mapping associated with each consecutive instruction. An execution totally orders successive instructions executed by a processor.

Similarly, we assume the existence of a formal definition for a correct execution by memory. A memory operation consists of three events:

(1) receives a memory *request* message;

(2) processes the request, possibly modifying the memory content; and

(3) sends a *reply* message (we assume that all accesses generate replies; the reply is an acknowledgment for accesses that do not return values).

We have M.receive.request → M.process.request → M.send.reply; memory operations are executed atomically.

A correct execution of the system must respect *data dependencies*: if instruction u precedes instruction v, and both access the same memory location, then the access on behalf of u must occur before the access on behalf of v. This correctness condition does not occur explicitly in our definitions; it pertains neither to the processor nor to the memory, but to their interaction. We shall show that it implicitly follows from the correctness requirements of the subsystems.

Let → be a partial order defined by a correct execution of the system consisting of processor and memory, and let $\in$ be the relation induced on instructions. Let u and v be two processor instructions that access the same memory location such that one of the instructions is a write and v follows u. We have $u \in v$. Assume, by contradiction, that the

memory access on behalf of v is executed before the memory access on behalf of u, i.e

$$M.process.request.v \rightarrow M.process.request.u \ .$$

Since, by the ordering of events of an operation, we know that

$$P.send.request.v \rightarrow M.receive.request.v \rightarrow M.process.request.v$$

and

$$M.process.request.u \rightarrow M.send.reply.u \rightarrow P.receive.reply.u \ ,$$

we get the ordering:

$$P.send.request.v \rightarrow M.receive.request.v \rightarrow M.process.request.v$$

$$\rightarrow M.process.request.u \rightarrow M.send.reply.u \rightarrow P.receive.reply.u$$

so that $v \in u$, and $\rightarrow$ cannot be extended to relation that induces a partial order on processor instructions.

A correct implementation of the processor/memory system must ensure that memory accesses are executed in an order consistent with the order instructions are issued, whenever there is a memory access conflict. Thus, the outcome of a (correct) execution is as if the instructions were executed serially.

### 3.4. Multiprocessors

We wish to extend these definitions to a shared memory multiprocessor. Such a machine consists of several processors and several shared memory modules. Each processor and memory module is defined as in the previous example. We assume the existence of a formal definition for the correct execution for a processor, and of correct execution for a memory module.

The correctness of the entire system is derived as previously: an execution relation $\rightarrow$ is correct if it can be extended to a relation, $\Rightarrow$ , that correctly orders events at each processor and at each memory module. If the execution is correct then the atomic events can be serially ordered so that events pertaining to the same processor instruction are contiguous. The outcome of the execution is as if the instructions were executed serially, with all events of one instruction terminating before any event of then next instruction starts, so that for each processor the subsequence of events of this processor is a valid execution for

the processor. This is the "sequential consistency principle" stated by Lamport [La1]. It implies that we can view a multiprocessor as a system of sequential processes communicating via shared variables, where each instruction is an atomic operation [LyF]; access to (shared) memory is perceived to occur simultaneously with the execution by the processor of the instruction that generates the access.

### 3.5. Asynchronous Memory Access

The sequential consistency principle can be enforced in hardware either by using a central controller for memory accesses [La1] or by requiring each processor to wait for an acknowledgement after each shared memory access (before beginning to process the next shared memory access). Both choices, however, severely limit the performance of a large scale parallel processor. A central controller becomes a serial bottleneck when there are a large number of processors. The network latency time is long (as compared to the basic instruction cycle time of each processor) in a shared memory machine with a large number of processors and/or memory modules. This latency time overhead can be mitigated by allowing the processor to continue processing before receiving an acknowledgment. For example, the NYU Ultracomputer and RP3 hardware allow the pipelining of shared memory accesses from the processors.

These machines present the user with a shared memory multiprocessor architecture with the following types of atomic events:

(1) Execution of a local instruction, i.e. instructions that involve only local stores; and

(2) Execution of events comprising a shared memory access operation (an RMW operation). We assume that each such operation involves only one shared memory module and consists of three atomic events:

SEND - a request message is issued by the processor.

ACCESS - the request message is consumed by a memory module, the request is executed, and a reply message is generated.

RECEIVE - the reply message is consumed by the processor.

The three components of the same shared memory access operation are ordered

$$\text{SEND} \rightarrow \text{ACCESS} \rightarrow \text{RECEIVE} .$$

The control logic of each processor may impose constraints on the sequencing of the events executed by the processor. However, it does not necessarily wait for a reply from a shared memory access before proceeding with another event.

We call a machine with such an architecture a *Multiprocessor with Asynchronous Shared Memory* or *MASM*. A MASM architecture does not necessarily fulfill the sequential consistency principle, i.e. is not "correct" according to the usual definitions; however, it can implement a sequentially consistent multiprocessor. The sequential consistency principle is enforced by a software solution, involving compile time analysis of the global code, that specifies constraints on the pipelining. These constraints are enforced by the control logic of each processor. For example, the NYU and RP3 software distinguishes between "private" variables, "shared" read-only variables and "shared" read/write variables (all of which can be stored in shared memory), and prohibits the pipelining of accesses to variables of the latter type. Shasha and Snir [SS] propose a more elaborate analysis based on compile time detection of data dependencies; this analysis is used to define "delay" pairs, i.e. pairs of memory accesses at the same processor such that the first access must complete before the second starts.

The last definition did not mention the communication medium between processors and memories. We assume that this interconnection network is "invisible"; its state is not observed by the user. Note, too, that we assume asynchronous processor to memory communication; it is only the relative order of events that affect the result, not the absolute time of their execution. Our formalism does not encompass synchronous communications, or time-out mechanisms.

## 4. Combining Mechanism

There have been many proposals for the architecture for parallel processors. The main issue is how to interconnect the processors so that they may communicate efficiently. While shared bus type architectures are well suited for interconnecting dozens of processors and memory modules, multistaged interconnection networks appear to be required for larger scaled parallel machines. We first describe our assumptions concerning the interconnection network and then give a general technique for "combining" common shared memory requests. We show that this implementation is correct in the sense described in the previous section.

## 4.1. Processor to Memory Connection

We assume a MASM architecture as defined in §3.5, and for the sake of definiteness, make the following additional assumptions:

(1) The processors communicate with shared memory modules via a multistaged interconnection network. The network is packet switched. It may be either multistage or recirculating.

(2) A reply message is sent back on the same path followed by the request message. This condition is trivially satisfied for multistage networks that have a unique path connecting each processor to each memory module. It is easy to enforce the condition in any network: A message can construct as it travels through the network a header describing its path; this header is used to route the reply in the reverse direction [GGK].

These assumptions are also made in the NYU Ultracomputer [GGK] and IBM's RP3 machine [PBH].

## 4.2. How to Combine Requests

We assume that memory accesses are RMW operations. A memory request message has the form <id,addr,f>, where id is an identifier that uniquely identifies the request, *addr* is a reference (address) to a memory location,[2] and *f* is (the encoding of) a mapping. When this message reaches memory, @addr, the contents of location addr, is replaced by f(@addr), and a message <id,@addr> containing the *original* contents of location addr is returned.

Suppose that two request messages of the form $<id_1,addr,f>$ and $<id_2,addr,g>$ meet at the same switch. These two messages have the same destination and thus conflict. We propose *combining* these two messages into a single message. This is done as follows:

(1) The switch stores $id_1$, $id_2$, and *f* and forwards the message $<id_1,addr,fog>$, where fog is (an encoding of) the composition[3] of *f* and g.

(2) When a reply message $<id_1,val>$ to this composed request reaches the switch, the stored information is retrieved by matching the id's; a message $<id_1,val>$ is

[2] The address may be part of the identifier. Thus, if each processor has at most one outstanding request to each address, then the processor number can be used as identifier.

forwarded as a reply to the first request $<id_1,addr,f>$, and a message $<id_2,f(val)>$ is forwarded as a reply to the second request $<id_2,addr,g>$.

Assume that the combined request $<id_1,addr,fog>$ is not further combined in the network. Then $<id_1,@addr>$ is returned as a reply, and the value @addr is replaced in memory by g(f(@addr)). At the switch the reply $<id_1,@addr>$ is forwarded (back) to the first request, and the reply $<id_2,f(@addr)>$ is forwarded (back) to the second request. This is illustrated in Figure 1. The final effect is as if the first request was executed, returning the value @addr and replacing it in memory with f(@addr), and then the second request was executed, returning the value f(@addr) and replacing it with g(f(@addr)). Combining is transparent: the operations executed by the processors and the final memory content are the same as would occur without combining.

### 4.3. Correctness of Combining Mechanism

We now show that this implementation is correct: The observable behavior in a computation of a combining machine is a behavior that could be observed in a computation of a noncombining machine. Note that the reverse is not necessarily true: There are sequences of observable events that occur in a noncombining machine but can not occur in a combining one. (We follow what Lamport calls the "restrictive" approach to specification [La3].)

Our implementation does not change the set of operations executed by the processors; it is transparent to the processor logic. It may reduce the number of ACCESS operations that are executed; however, the memory state that occurs after the execution of an ACCESS operation in the combining machine could occur in some valid computation of the noncombining machine (after the execution of some sequence of ACCESS operations). In other words, for each sequence of operations in a combining machine there exists a sequence of operations in a noncombining machine that is equivalent in the following sense:

(1) The same operations, in the same order, are executed by the processors in either machine.

(2) The value of each RECEIVE message is the same in both machines.

(3) The final value of each shared memory location is the same in both machines.

[3] We use fog(x) to denote g(f(x)).

Note that this does not imply that the combining machine satisfies the sequential consistency principle. It only implies that the combining machine is a correct implementation of a MASM architecture. Any mechanism that can be used by the processors of a noncombining machine to enforce sequential consistency will achieve the same goal on a combining machine.

In general, a combined request can be further combined. An inductive proof is needed to show that the final outcome is correct. In a noncombining network each SEND is associated with one ACCESS and one RECEIVE; in a combining network each SEND is associated with one RECEIVE, but several SEND operations may result in one (combined) ACCESS.

Each memory request message in the network is associated with a sequence of memory request messages issued by processors. A memory request issued by a processor *represents* itself; if memory request A was obtained by combining B with C, where B represents requests $b_1, \cdots, b_i$ and C represents requests $c_1, \cdots, c_j$, then we say A *represents* requests $b_1, \cdots, b_i, c_1, \cdots, c_j$.

**Lemma**: Consider a combining machine as in §4.2. Let A = <id,addr,f> be a memory request message, representing requests $a_1 = <id_1,addr,f_1>, \cdots, a_n = <id_n,addr,f_n>$. Let $a'_i$ be the reply message associated with $a_i$, i.e. the reply message $<id_i,val>$ received by the processor that issued $a_i$. Then

(1) $f = f_1 o \cdots o f_n$;

(2) The values returned by all of the $a'_i$ are the same as would be returned if the memory accesses associated with requests $a_1, \cdots, a_n$ in a noncombining network were executed consecutively.

(3) If request A reaches memory without being combined, the value stored at location addr after execution of request A is the same as the final value stored at location addr after consecutively executing the memory accesses associated with $a_1, \cdots, a_n$ in a noncombining network.

**Proof:** The lemma is proven by induction on the number of requests represented by a memory access message. It is trivial for a message that represents one request. Next, assume that the lemma is true for messages representing less than n requests, and assume

that A is obtained by combining B and C, where B represents r requests and C represents $n-r$ requests ($1 \leq r < n$). Let $B = \langle id^B, addr, g \rangle$ and $C = \langle id^C, addr, h \rangle$, so that $A = \langle id^B, addr, goh \rangle$. Then message A generates a reply $\langle id^B, val \rangle$, which will also be the reply to request B; request C generates the reply $\langle id^C, g(val) \rangle$. If A reaches memory then val = @addr and the new value in memory is h(g(@addr)).

Let $b_1 = \langle id_1^b, addr, g_1 \rangle, \cdots, b_r = \langle id_r^b, addr, g_r \rangle$ be the sequence of requests B represents; similarly, let $c_1 = \langle id_1^c, addr, h_1 \rangle, \cdots, c_{n-r} = \langle id_{n-r}^c, addr, h_{n-r} \rangle$ be the sequence of requests C represents. Let $b'_i$ and $c'_i$ be the reply messages associated with the respective requests. By the inductive assertion, $g = g_1 o \cdots o g_r$ and $h = h_1 o \cdots o h_{n-r}$; the messages $b'_i$ return the values val, $g_1(val)$, ..., $g_{r-1}(\cdots(g_1(val))\cdots)$; the messages $c'_i$ return the values g(val), $h_1(g(val))$, ..., $h_{n-r-1}(\cdots(h_1(g(val)))\cdots)$. It follows that the values returned, and the new memory value when A reaches memory are as if the memory accesses associated with $b_1, \cdots, b_r, c_1, \cdots, c_{n-r}$ in a noncombining network were successively executed in this order. This proves the lemma. □

**Theorem:** The implementation of shared memory access by a combining network is correct.

**Proof:** The previous lemma clearly implies the theorem: Indeed, let $a_1, \cdots, a_n$ be a serialization of the events in an execution of a machine with a combining network. Replace each ACCESS event $a_i$ by the sequence of ACCESS events associated (in a machine with noncombining network) with the requests represented by the message that generated $a_i$. Then all events occurring at processors appear in the same order in both sequences; the RECEIVE messages return the same values; and the shared memory state after the execution of an ACCESS event in the first sequence is identical with the memory state after execution of the corresponding sequence of ACCESS events in the second sequence. □

## 5. Applications

Suppose one intends to combine RMW requests with mappings from some family $\Phi$ of transformations. Composition can generate any mapping in the semigroup[4] $\overline{\Phi}$ spanned by

$\Phi$. We need to have an encoding for the mappings in $\overline{\Phi}$ so that

(1) the computer representations of mappings from $\overline{\Phi}$ have reasonable size;

(2) the encoding of fog can be easily computed from the encoding of *f* and the encoding of g; and

(3) f(a) can be easily computed from the computer representations of *f* and a.

We shall not give a formal definitions of "reasonable" or "easily computed", as these are application dependent; we have in mind encodings that use a small constant number of words, and computations that require few machine cycles. We say that $\Phi$ is *tractable* if it fulfills these conditions.

## 5.1. Loads and Stores

Recall that a load from variable X is equivalent to RMW(X,**id**), where **id** is the identity mapping, and a store (actually a swap) of value v to variable X is equivalent to RMW(X,$I_v$), where $I_v$ is the mapping that has constant value v. The set of mappings $\{I_v\} \bigcup \{\mathbf{id}\}$ is a semigroup, and composition is easily computed. A mapping from this semigroup is represented by one computer word and one opcode bit. The composition yields the expected results:

- A load followed by a load combine into a load.
- A load followed by a store combine into a store (the value fetched is returned to the load).
- A store followed by a load combine into a store (the value being stored is returned to the load).
- A store followed by a store combine into a store of the second value.

One need not transmit the value returned by a store request, as this is of no interest; an acknowledgment suffices. A combined request needs to return a value only if the first atomic request in it is a load operation. One can avoid returning values in the other cases by tagging these instructions. Then, with the possible exception of these extra tag bits, combining never generates extra traffic; often it will decrease it significantly.

---

[4] A semigroup is a set closed under an associative operation, which in this case is map composition.

Note that, in general, the order of combined requests is arbitrary and can be reversed. This can be used to further simplify combining. For example, if the network always chooses to effect a store before a load whenever two such requests are combined, then a store never needs to return a value.

The situation of a store combined with a load, suggests a slight improvement in performance by satisfying the load immediately. That is, the store would be forwarded to the memory module and its value will also be returned, as soon as possible, back to the processors that issued the load. A computation on such machine is still equivalent to a computation on a machine with noncombining network, where local operations SEND, ACCESS, and RECEIVE are atomic events. However, it is no longer true that ACCESS → RECEIVE; a processor may get a reply to a load request long before the value returned is actually stored in memory. This departure from the MASM model may lead to an incorrect behavior; in particular, constraints on the scheduling of events at each processor can not enforce sequential consistency.

If $\Phi$ is a semigroup of mappings, then $\Phi \bigcup \{I_v\} \bigcup \{id\}$ is a semigroup too. We have

$$f \circ id = id \circ f = f ,$$

$$f \circ I_v = I_v , \quad \text{and}$$

$$I_v \circ f = I_{f(v)} .$$

Thus, if $\Phi$ is tractable, then $\Phi \bigcup \{I_v\} \bigcup \{id\}$ is tractable. In other words, it is always possible to add load and store operations to a family of RMW operations, and combine them all, without greatly increasing the complexity of the system.

Our discussion has assumed that stores and loads always affect an entire memory cell (word of memory). If we assume a word-addressable machine, say with four byte words, then combination of store operations that affect only bytes or half-words will require introducing store operations that affect any subset of bytes in a word. At a higher level, if one combines atomic stores that affect components of a structured variable then one needs to support stores that affect an arbitrary subset of the components of this variable.

## 5.2. Associative Operations

Let $\theta$ be an associative operation. Then *fetch-and*-$\theta(X,a)$ is equivalent to $RMW(X,\theta_a)$, where $\theta_a(x) = x\theta a$. The function *fetch-and*-$\theta(X,a)$ corresponds to the indivisible execution of the following code.

```
function fetch-and- θ(X,a)
   begin
      temp ← X;
      X ← Xθa;
      return(temp)
   end
```

A fetch-and-$\theta(X,a)$ followed by fetch-and-$\theta(X,b)$ can combine into fetch-and-$\theta(X,a\theta b)$, since

$$\begin{aligned} \theta_a \circ \theta_b(x) &= \theta_b(\theta_a(x)) \\ &= (x\theta a)\theta b \\ &= x\theta(a\theta b) \qquad \text{(since } \theta \text{ is associative)} \\ &= \theta_{a\theta b}(x) \ . \end{aligned}$$

Thus, the semigroup $\{\theta_a\}$ is tractable whenever $\theta$ is easy to compute.

Perhaps the most important fetch-and-$\theta$ primitive for large-scale shared memory machines is the fetch-and-add, which was discussed earlier. The mapping can be represented by one computer word (the addend). Two other potentially useful fetch-and-$\theta$ primitives are *fetch-and-OR*, where *OR* is Boolean addition, and *fetch-and-min*. Fetch-and-OR(X,1) is the test-and-set operation. Fetch-and-min is useful for allocation with priorities.

## 5.3. Boolean Operations

The sixteen Boolean operations can also be combined, despite the fact that some of them are not even associative operations. Moreover, each of the operations can be applied to bit vectors, of one word size. We will first consider the unary Boolean operations.

Let $\Phi$ be the set of four Boolean functions on one variable, **0**, **1**, **x**, and $\overline{\mathbf{x}}$. The associated RMW operations are *test-and-clear*, *test-and-set*, *load*, and *test-and-complement*. The four functions in $\Phi$ can be represented by two bits, and can be composed using the following 4×4 table.

| | load | clear | set | comp |
|---|---|---|---|---|
| load | load | clear | set | comp |
| clear | clear | clear | set | set |
| set | set | clear | set | clear |
| comp | comp | clear | set | load |

The function compositions can be computed in hardware with few gates. Thus $\Phi$ is a tractable semigroup.

As a result all sixteen operations fetch-and-$\theta$, where $\theta$ is a binary Boolean operation, can be combined. The reason is that the value of the second variable is fixed to a constant (0 or 1) when a request is issued, and every Boolean operation on two variables with one of the variables fixed is equivalent to some Boolean operation on one variable. For example, fetch-and-AND(X,a) is a load when $a=1$, and is a test-and-clear when $a=0$.

This result can be extended to Boolean operations on bit vectors. Mappings on bit vectors of length n are represented by 2n bits. Such operations are useful to support multiple locking.

## 5.4. Arithmetic Operations

Let $\Psi$ be the set of arithmetic operations addition, subtraction, multiplication, and division. We also put into $\Psi$ the reverses of the two noncommutative operations: reverse subtraction of a and b is b−a, and reverse division of a and b is b/a. We wish to support and combine all the operations of the form *fetch-and-*$\psi$, where $\psi \in \Psi$. In order to do that, we need to support and combine the operations $RMW(X,\psi_a)$, where $\psi \in \Psi$, and $\psi_a(X) = X\psi a$. The semigroup spanned by the set of mappings $\{\psi_a : \psi \in \Psi\}$ consists of the Moebius functions: These are the functions of the form

$$x \rightarrow \frac{ax+b}{cx+d},$$

where a, b, c, and d are constants, and either $c \neq 0$ or $d \neq 0$.

We represent the function

$$x \rightarrow \frac{ax+b}{cx+d}$$

by the 2x2 matrix of coefficients

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

If $f_A$ is the Moebius function represented by the matrix A, then

$$f_A \circ f_B = f_{BA}.$$

Thus, a function is represented by four coefficients, and two functions are composed by multiplying two 2x2 matrices.

We can now efficiently support all assignments of the form $x \leftarrow x \theta c$, where $\theta$ is an arbitrary arithmetic operation, and c is a constant or a private variable. These assignments will be executed atomically, while still being combined in the network. Such assignments form a large part of the machine code in typical applications.

If one wishes to support only addition and multiplication, then it is sufficient to consider functions of the form

$$x \rightarrow ax+b\ ,$$

which can be represented using only two coefficients. Combining two such mappings requires two multiplications and one addition.

Hardware arithmetic operations are not associative. Use of the associativity law may change occurrences of overflows in integer arithmetic, and may change occurrences of overflows, underflows, and rounding errors in floating point arithmetic. As our combining mechanism relies on associativity, the arithmetic may not produce the same results as would the serial order of the operations. Furthermore, the transformations used are not numerically stable when division occurs; they are numerically stable when divisions are left out. In that respect, our combining mechanism suffers from the same shortcomings as compiler optimization techniques that use transformations based on algebraic identities.

It is possible to obtain an accurate combining mechanism for fixed point operations, not including division, by adding one extra bit to the intermediate values, thereby increasing the range by a factor of two. If an overflow occurs in that increased range then an overflow would have occurred in the serial execution of the operations in the restricted

range. A similar technique of using guard bits will keep rounding errors under control when floating point operations not involving division are combined.

## 5.5. Full-Empty Bits

Accesses to shared variables can be synchronized using memory tags. For example, the HEP computer uses a *full-empty* bit at each shared memory word [Sm]. These bits can be used to synchronize accesses in a producer consumer fashion. Writing may be conditional on the location being empty; a successful write sets the (full-empty) bit. Reading may be conditional on the location being full; a successful read may clear the (full-empty) bit.

A load operation has the same effect in memory as the corresponding conditional load operation. We may therefore assume that load operations are always executed unconditionally: a processor can check the value of the full-empty bit returned by the load instruction to determine if it was successful. A conditional store instruction that fails returns a negative acknowledgement; the processor may resend it later.

In order to implement this synchronization mechanism, consider the four memory access instructions (which are defined formally below) that form the basis of those in tagged memory architectures: load, load-and-clear, store-and-set, and store-if-clear-and-set.

Let the pair (X,flag) represent the variable X and its associated full-empty bit flag. Temporarily assume that stores are actually implemented as swaps, i.e. they return the old value. In order to implement the instruction set as RMW instructions, one needs four types of mappings.

(1) The identity mapping for *load*: (X,flag) → (X,flag).

(2) The mapping for *load-and-clear*: (X,flag) → (X,0).

(3) The mapping for *store-and-set*: (X,flag) → (v,1).

(4) The mapping for *store-if-clear-and-set*:

$$(X,\text{flag}) \rightarrow \begin{cases} (v,1) & \text{if flag} = 0 \\ (X,1) & \text{if flag} = 1 \end{cases}$$

To close this set of mappings under composition, two more mappings must be included:

(5) The mapping (X,flag) → (v,0) is a *store-and-clear*. It implements a store-and-set followed by a load-and-clear.

(6) The mapping

$$(X,\text{flag}) \rightarrow \begin{cases} (v,0) & \text{if flag} = 0 \\ (X,0) & \text{if flag} = 1 \end{cases}$$

is a *store-if-clear-and-clear*. It implements a store-if-clear-and-set followed by a load-and-clear.

These requests can now be combined. The combine logic is simple. Each of the six types of instructions can be encoded by a short opcode, an address, and optionally a data word.

A store request carries one data value. A reply to a request needs to carry a data value only if the request is a load or a combined store that contains a simple load instruction. If these store instructions are handled specially, then the number of data values transmitted through a combining network will never exceed the number that would have been transmitted in an uncombining network.

There is a problem if the instruction set includes a standard store instruction, i.e. one that does not change the full-empty bit. If a store followed by a store-if-clear-and-set are to combine, it cannot be determined *a priori* which store will actually be executed. One solution is to forward both store values. A better solution is simply to reverse the order of the requests (to be the store-if-clear-and-set followed by the store). These can be forwarded as a store-and-set instruction.

Reversing the order does not always help. For example, if the operations store-if-clear and store-if-set are combined, both store values have to be forwarded. As we will see in the next section in a much more general context, even if we include all types of full-empty instructions, no request will ever have to carry more than two store values.

We assumed in this section a *busy-waiting* model for synchronization: an operation that fails returns a negative acknowledgement; the processor may retry later. An alternative mechanism is to *queue* a request at memory until it is executable. This decreases the

network traffic. However, unless some time-out mechanism is available at the memory controller, the hardware may deadlock.

Assume the two operations load-and-clear-if-set and store-and-set-if-clear are used to access memory in a queueing system. Memory accesses at a location are executed in a sequence of alternating loads and stores. Thus, a set of i load and j stores can be combined into $|i-j|+1$ operations: stores are combined with loads, with the excess of loads or stores staying uncombined. While combining is not guaranteed to reduce traffic in the worst case, one can expect it will do so in the average case.

### 5.6. Data-Level Synchronization

One can have more than two possible states (full and empty), and operations other than read and write on data. In a general *data-level synchronization scheme*, we have a semigroup $\Phi$ of mappings representing the RMW operations that can be executed, and a set S of states. Each variable is tagged by its state. The execution of an operation on a variable is conditional on its being in a suitable state; the operation also changes the variable's state.

This mechanism can be represented by an automaton $A = <\Phi,S,\delta>$, where $\delta:S\times\Phi \rightarrow S$ is the state transition function. Assume that variable X is in state s, and an RMW(X,f) instruction is issued. If $\delta(s,f) = \epsilon$ (i.e. undefined) the instruction fails, and a negative acknowledgement is returned. Otherwise, RMW(X,f) is executed, and the new state of X is set to $\delta(s,f)$. Define the mapping f′ by

$$f'(X,s) = \begin{cases} (\, f(X)\, ,\, \delta(s,\, f)) & \text{if } \delta(s,\, f) \neq \epsilon \\ (X,\, s) & \text{otherwise} \end{cases}$$

Then the execution of the instruction RMW(X,f) under the control of the automaton A is equivalent to the execution of the instruction RMW((X,s),f′).

Consider now the case where the operations executed are stores and loads. The basic instructions are then

(1) load(X,S,$\delta$): Load from X if state s is in S and change state to $\delta(s)$.

(2) store(X,v,S,$\delta$): Store the value v into X if state s is in S and change state to $\delta(s)$.

For uniformity, we represent a load by the tuple $(X,\Omega,S,\delta)$, where the special value $\Omega$ represents the fact that no store is executed. A combined request then has the form $<X,$

$(v_1,S_1,\delta_1)$, ..., $(v_k,S_k,\delta_k)>$, where the $S_i$ are disjoint sets of states. The meaning of this instruction is: if state s is in $S_i$ then store $v_i$ (or store nothing if $v_i = \Omega$) and change to state $\delta_i(s)$. If s is not in any $S_i$, then the instruction fails.

A combined instruction that represents k atomic store instructions carries at most k store values. Also, a combined instruction never carries more than $|S|$ store values, where $|S|$ is the number of states of the controlling automaton A. This is in general the best possible bound: if there is an instruction *store-if-state*=s for each state s of A, then a combined store may have to carry a distinct store value for each state. This is tractable when the number of states is small, such as when a full-empty bit is used; it is not tractable when the number of states is large. For example, the synchronization primitives defined by Zhu and Yew [ZY] for the Cedar machine at the University of Illinois and by Pier and Gajski [GP] use full word tags. With m bit tags, there are $2^m$ possible states, and $2^m$ is the best possible uniform bound on the number of store values in a combined request.

Memory accesses controlled by a regular automaton can be used to support *simple path expressions* [CH]. Path expressions are used to synchronize access to shared objects. For each such object there is a set of possible operations on it. A regular expression over the alphabet consisting of these operations defines the language of legal sequences of operation applications on each object.

A deterministic automaton corresponding to the path expression is built. Each object is represented by a variable in memory, to which access is protected by this automaton. Each execution of a protected operation is preceded by an access to that variable that performs the corresponding automaton transition. Then the executions of the operations are sequenced according to the path expression. The mechanism suggested in this section allows an efficient implementation of such a system.

## 6. Rmw and Parallel Prefix

This section shows the relationship between the combining mechanism presented in this paper with a well known computational problem, prefix computation. The combining logic turns out to be an asynchronous version of a well known parallel synchronous algorithm. This sheds further light on performance aspects of combining.

Consider successive execution of the operations RMW(S,$f_1$), ..., RMW(S,$f_n$). These operations return the values S, $f_1(S)$, ..., $f_{n-1}(\cdots(f_1(S))\cdots)$; the value $f_n(\cdots(f_1(S))\cdots)$ is stored in memory. Thus, execution of these instructions amounts to the computation of S, $f_1(S)$, ..., $f_n(\cdots(f_1(S))\cdots)$ or, equivalently, to the computation of $I_S$, $I_S$o $f_1$, ..., $I_S$o $f_1$o $\cdots$ o $f_n$. This is a particular instance of the *prefix computation* problem [LaF]: given $x_1$, ..., $x_n$, compute $x_1$, $x_1$*$x_2$, ..., $x_1$*$\cdots$*$x_n$, where the operation * is an arbitrary associative operation. In our case, * is map composition.

Prefix computation when solved in parallel is known as *parallel prefix*. The memory access mechanism proposed in this paper provides in fact a parallel solution to the prefix computation problem. The computations are performed on the nodes of a tree in the interconnection network that connects the processors to one memory module. In a multistage network, in which processors have at most one outstanding request to each memory location, this is a physical tree, which is a subgraph of the network. In other cases this is a virtual tree: operations pertaining to distinct levels in the tree are executed at the same node of the network.

The problem solved by the combining network differs from parallel prefix in that the order of the elements combined (with the exception of the first) is arbitrary. By ordering the operations correctly, one obtains a distributed, asynchronous network that solves the parallel prefix problem.

The computation is performed on a network of processes connected as a (not necessarily complete) binary tree with n leaves. The inputs are stored at the n leaves of a binary tree, which corresponds to the processors of the parallel computer. The root of the tree has one parent, called *superoot*; it corresponds to the memory module that contains the variable accessed; the internal nodes of the tree correspond to the combining switches in the processor to memory interconnection network. We describe below in CSP notation [Ho] the different types of processes.

**Leaf Process**

```
[Leaf:: val;
   parent ! val;
   parent ? val
]
```

**Internal Node Process**

```
[Node:: lval, rval, pval;
   left_child ? lval;
   right_child ? rval;
   parent ! lval*rval;
   parent ? pval;
   left_child ! pval;
   right_child ! pval*lval
]
```

**Superoot Process**

```
[Superoot:: val;
   child ? val;
   child ! id
]
```

Let $val_i$ be the initial value at the $i$-th leaf. At the end of the computation the value at the $i$-th leaf equals to $val_1 * \cdots * val_{i-1}$; the value at the superoot equals to $val_1 * \cdots * val_n$.

If the tree is complete, then the operations performed by this tree are exactly the same operations performed by the Ladner-Fisher parallel prefix network [LaF]. The global clock synchronization used by their algorithm is replaced by local data-flow synchronization. Each internal node performs two multiplications, of which $\lceil \lg n \rceil$ are trivial. Thus, $2n-2-\lceil \lg n \rceil$ nontrivial multiplications are done. The algorithm can be implemented to run in $2\lceil \lg n \rceil - 2$ multiplication cycles, when globally synchronized.

## 7. Conclusion

This paper provides and exemplifies a formal method for reasoning about the correctness of parallel computer architectures. It provides the theoretical underpinnings of the combining mechanism used by the NYU Ultracomputer and RP3. It presents a general formulation of RMW operations and a general mechanism to efficiently support such operations.

A significant amount of supplementary hardware is required to combine RMW operations. Each switch needs logic that is able to compute mapping compositions and mapping applications; extra logic is also required at the memory module. The switches also need an associative store to store information on combined requests.

The need for associative retrieval at the switches can be avoided at the expense of more expensive labeling schemes. An implementation of an efficient switch that supports

combining of fetch-and-add requests is described in [DKS], [DKSS]. This switch has been partially implemented. The same scheme can be used for other RMW operations.

Note that one can use combining logic that detects only part of the combinable pairs. Memory accesses are correctly performed even with partial combining, or no combining at all. Thus, different cost-performance tradeoffs are possible.

Combining or partial combining can be used on a wide variety of interconnection networks. The only major restriction is that requests must return via the same route (although in the reverse direction). Thus, the mechanisms described in this paper can be easily adopted for use by direct connection machines, such as the cosmic cube [Se], where the processors themselves act like network switches and the local memories at each node are all view as part of a distributed, shared memory.

## REFERENCES


[ABB] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Architecture of the IBM System/360, *IBM J. Research and Development*, pp. 87-101, April 1964.

[CH] Campbell, R. H. and A. N. Haberman, The Specification of Process Synchronization by Path Expressions, *Lecture Notes in Computer Science, vol. 16*, Springer-Verlag, pp. 89-102, Aug. 1976.

[DKS] Dickey, S., R. Kenner, and M. Snir, An Implementation of a Combining Network for the NYU Ultracomputer, Ultracomputer Note #93, NYU, Jan. 1986.

[DKSS] Dickey, S., R. Kenner, M. Snir, and J. Solworth, A VLSI Combining Network for the NYU Ultracomputer, *IEEE Proc. of the Intl. Conf. on Computer Design*, pp. 110-113, Oct. 1985.

[Di] Dijkstra, E. W., Hierarchical ordering of sequential processes, *Acta Informatica, vol. 1*, pp. 115-138, 1971.

[DGSS] Draughon, E., R. Grishman, J. Schwartz, and A. Stein, Programming Considerations for Parallel Computers, Rep. IMM 362, Courant Inst. of Mathematical Sciences, New York Univ., New York, N. Y., 1967.

[GGK] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, The NYU Ultracomputer -- Designing an MIMD Parallel Computer, *IEEE Transactions on Computers, vol. C-32*, pp. 75-89, 1984.

[GK] Gottlieb, A. and C. P. Kruskal, Coordinating Parallel Processors: A Partial Unification, *SIGARCH*, Oct. 1981.

[GLR] Gottlieb, A., B. D. Lubachevsky, and L. Rudolph, Efficient Techniques for Coordinating Sequential Processors, *ACM TOPLAS*, pp. 164-189, 1983.

[GP] Gajski, D. D. and J.-K. Peir, Essential Issues in Multiprocessor Systems, *IEEE Computer*, pp. 9-28, June 1985.

[Ho] Hoare, C. A. R., Communicating Sequential Processes, *CACM vol. 21*, pp. 666-677, Aug. 1978.

[La1] Lamport, L., How To Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers, vol. C-28*, pp. 690-691, 1979.

[La2] Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, pp. 558-565, July 1978.

[La3] Lamport, L., On Interprocess Communication, to appear in *Distributed Computing*.

[LaF] Ladner, R. and M. J. Fisher, Parallel Prefix Computations, *JACM vol. 27*, pp. 831-838, 1980.

[LyF] Lynch, N. and M. J. Fisher, On Describing the Behavior and Implementation of Distributed Systems, *Theoretical Computer Science vol. 13*, pp. 17-43, 1981.

[MM] Milne, G. and R. Milner, Concurrent Processes and their Syntax, *JACM vol. 26*, pp. 764-772, 1979.

[PG] Peir, J.-K. and Gajski D. D., Data Flow Execution of Fortran Loops, *Proc. 1st Intl. Conf. on Supercomputing Systems*, Dec. 16-20, 1985.

[PET] Peterson, J. and A. Silbershatz, *Operating System Concepts*, Addison-Wesley, 1983.

[PBH] Pfister, G. H., et al., The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, 1985 Intl. Conf. on Parallel Processing, pp. 764-772.

[PN] Pfister G. H., and A. Norton, 'Hot Spot' Contention and Combining in Multistage Interconnection Networks, IEEE Transactions on Computers vol. C-34, pp. 933-938, 1985.

[Ru] Rudolph L., Software Structures for Ultraparallel Computing, PhD Thesis, New York University, 1981.

[SBK] Sullivan, H., T. R. Bashkow, and D. Klappholz, A Large Scale, Homogeneous, Fully Distributed Parallel Machine, *The 4th Ann. Symp. on Computer Architecture*, pp. 105-134, 1977.

[Se] Seitz, C., The Cosmic Cube, *CACM vol. 28* pp. 22-33, Jan. 1985.

[Sm] Smith, B. J., Architectures and Applications of the HEP Multiprocessor Computer System, *Real-Time Signal Processing IV, SPIE,* pp. 241-248, 1981.

[SS] D. Shasha and M. Snir, Efficient and Correct Execution of Programs That Share Memory, Ultracomputer Tech. Rep., NYU, Feb. 1986.

[ZY] Zhu and Yew, A Synchronization Scheme and Its Applications for Large Scale Multiprocessor Systems, *Proc. Conf. on Distributed Computing Systems,* pp. 486-491, 1984.

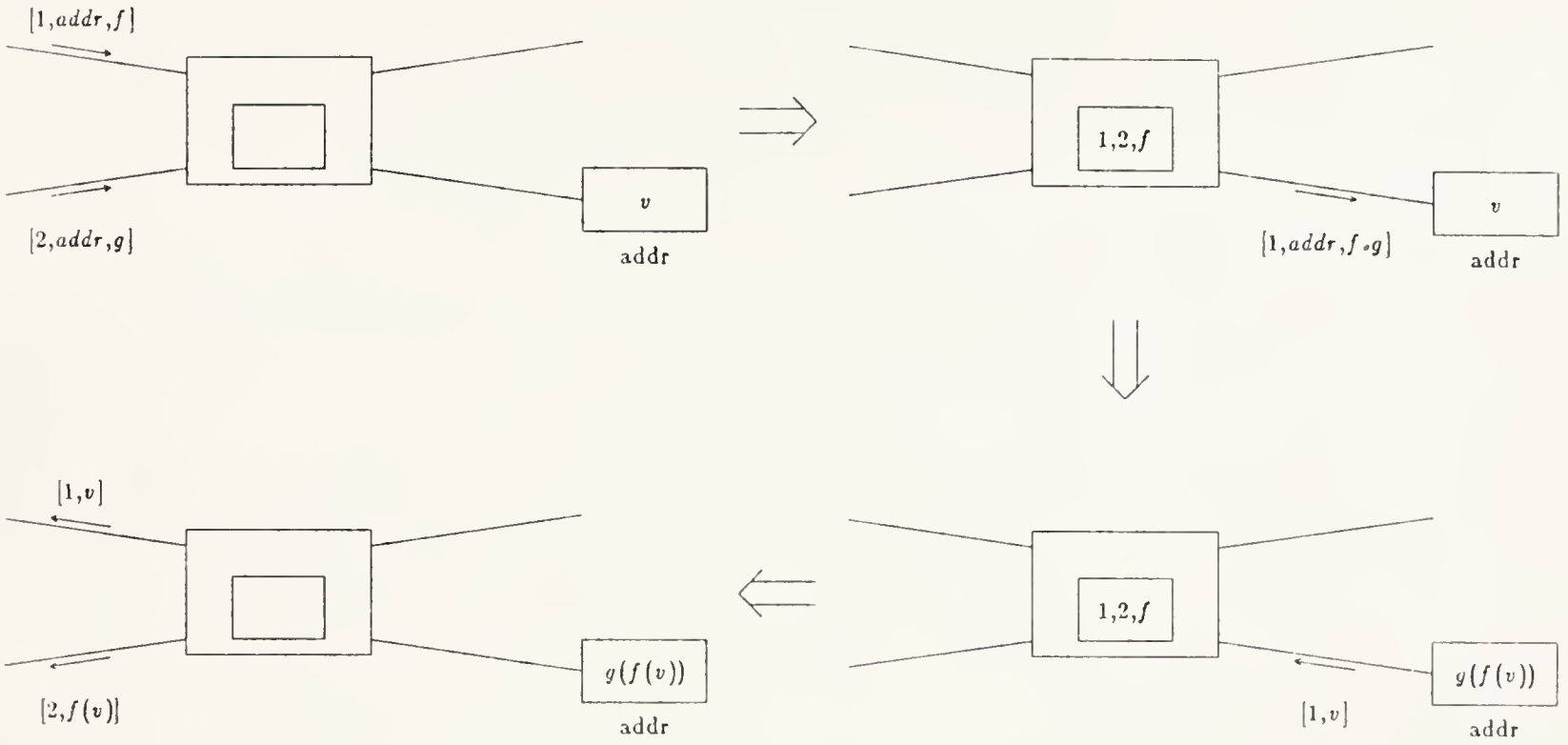

[1,addr,f]
[2,addr,g]
v
addr
1,2,f
[1,addr,f∘g]
v
addr
[1,v]
[2,f(v)]
g(f(v))
addr
1,2,f
[1,v]
g(f(v))
addr

Figure 1